Improving Motion Planning Algorithms by Efficient Nearest-Neighbor Searching

Anna Yershova, Steven M. LaValle

***Abstract*— The cost of nearest-neighbor calls is one of the bottlenecks in the performance of sampling-based motion planning algorithms. Therefore, it is crucial to develop efficient techniques for nearest-neighbor searching in configuration spaces arising in motion planning. In this paper we present and implement an algorithm for performing nearest-neighbor queries in Cartesian products of $\mathbb{R}$, S^1 and $\mathbb{R}P^3$, the most common topological spaces in the context of motion planning. Our approach extends the algorithm based on kd-trees, called ANN, developed by Arya and Mount for Euclidean spaces. We argue the correctness of the algorithm and illustrate substantial performance improvement over brute-force approach and several existing nearest-neighbor packages developed for general metric spaces. Our experimental results demonstrate a clear advantage of using the proposed method for both probabilistic roadmaps (PRMs) and Rapidly-exploring Random Trees (RRTs).**

***Index Terms*— Sampling-based motion planning, nearest-neighbor searching, kd-trees, configuration space, RRTs, PRMs.**

I. INTRODUCTION

NEAREST-neighbor searching is a fundamental problem in many applications, such as pattern recognition, statistics, and machine learning. It is also an important component in several path planning algorithms. Probabilistic roadmap (PRM) approaches [2], [17], build a graph of collision-free paths that attempts to capture the connectivity of the configuration space. The vertices represent configurations that are generated using random sampling, and attempts are made to connect each vertex to nearby vertices. Some roadmaps contain thousands of vertices, which can lead to substantial computation time for determining nearby vertices in some applications. Approaches based on Rapidly-exploring Random Trees (RRTs) [20], [22], [24] rely even more heavily on nearest neighbors. An RRT is a tree of paths that is grown incrementally. In each iteration, a random configuration is chosen, and the RRT vertex that is closest (with respect to a predefined metric) is selected for expansion. An attempt is made to connect the RRT vertex to the randomly-chosen state.

An approach that efficiently finds nearest neighbors can dramatically improve the performance of these path planners. Several packages exist, such as ANN ([26], U. of Maryland), Ranger (SUNY Stony Brook), which are designed for efficient nearest-neighbor generation in $\mathbb{R}^d$. These techniques, however,

Manuscript received January 20, 2002; revised November 18, 2002. This work was supported in part by NSF CAREER Award IRI-9875304, NSF ANI-0208891, and NSF IIS-0118146.

Anna Yershova (corresponding author) is with the Department of Computer Science, University of Illinois, Urbana, IL 61801 USA (email: yershova@cs.uiuc.edu).

Steven M. LaValle is with the Department of Computer Science, University of Illinois, Urbana, IL 61801 USA (email: lavalle@cs.uiuc.edu).

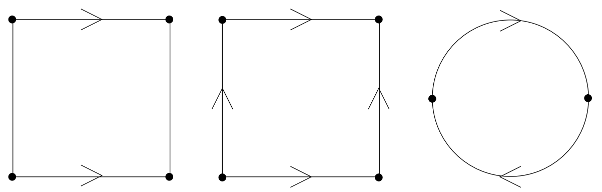


Fig. 1. Some 2D manifolds obtained by identifications of the boundary points of subsets of $\mathbb{R}^2$. Arrows on a pair of opposite edges indicate identification of the opposite points on the edges. If arrows are drawn in opposite directions, then there is a twist in the identification.

are developed uniquely for Euclidean spaces and cannot be applied directly to path planning algorithms because of the topologies of configuration spaces. The topologies that usually arise in the context of motion planning are Cartesian products of $\mathbb{R}$, S^1, and $\mathbb{R}P^3$, real projective space, for which metric information must be appropriately processed by any data structure that performs correct nearest-neighbor computations. Several other nearest-neighbor packages exist, such as sb(S) [9], and cover trees [7], that answer nearest-neighbor queries in general metric spaces. These packages use the metric function provided by the user as a “black box” for building a data structure based only on metric evaluations between the data points. Since any valid metric can be provided as the input, these methods are very general and usually introduce high computational overhead for Euclidean spaces and simple topological spaces that arise in motion planning.

Kd-trees [13], [28], [4] are well known for their good performance on Euclidean data sets. They usually outperform other approaches in practice, except in rare pathological cases. In this paper, we show how the kd-tree-based nearest-neighbor algorithm and part of the ANN package of Arya and Mount [26] can be extended to handle topologies arising in motion planning. The resulting method retains the performance benefits of kd-trees by introducing a very little computational overhead for handling the appropriate constraints induced by the metric and topology of the configuration space. First, we formulate the problem and describe the appropriate metric spaces in Section II. A literature overview of existing techniques for nearest-neighbor searching is covered in Section III. We then present our algorithm and prove the correctness of the approach in Section IV. We demonstrate the efficiency of the algorithm empirically in Section V. Our experiments show the performance improvement of the proposed algorithm over using linear-time naive nearest-neighbor computations, the sb(S) library, and the cover-tree library. The speedup is a few orders of magnitude in some cases. We also present experiments that show substantial performance improvement in the PRM and RRT methods applied to difficult path planning examples. We have implemented the proposed method as a software package publicly available at [29].

II. Problem Formulation

The configuration space, $\mathcal{C}$, which arises in motion planning problems is usually a non-Euclidean manifold or a collection of manifolds. A 2D rigid body freely translating and rotating in the plane has the configuration space $\mathcal{C} = \mathbb{R}^2 \times S^1$, in which circle S^1 represents the 2D rotations. 3D rigid body rotations lead to three-dimensional real projective space configurations, $\mathbb{R}P^3$. Toroidal manifolds arise as the configuration spaces of revolute joints of a manipulator. In the case of multiple bodies the resulting configuration space is a Cartesian product of the copies of $\mathbb{R}$, S^1, and P^3. When several of the joints of a manipulator form closed loops, the configuration space is usually a collection of submanifolds of one of the above configuration spaces (see [23] for more details).

Many of these d-dimensional configuration spaces can be represented by defining a subset of $\mathbb{R}^d$, and identifying appropriate pairs of boundary points to obtain the desired topology. For example, several two-dimensional manifolds can be obtained by identifying points on the unit square or unit circle in the plane, as shown in Figure 1. When motion planning is performed on such configuration spaces, an appropriate metric needs to be defined, and the search for nearest neighbors must be performed with respect to the metric and topology of the space. In this section we describe the metrics that are used for the most common configuration spaces in motion planning, and we formulate the nearest-neighbor problem for these spaces.

A. Common Metric Spaces

Throughout this paper we consider the following metric spaces.

1) Euclidean one-space: it arises from rigid translations, and is represented by $(0,1) \subset \mathbb{R}$. The metric for two points $p, q \in \mathbb{R}^1$ is defined as

$$\mathrm{dist}_{\mathbb{R}}(q,p) = |q-p|.$$

2) Circle, S^1: it can be represented by $S^1 = [0,1]/0 \sim 1$, a unit interval with identified endpoints. This configuration space arises from 2D rigid rotations. The metric for two points $p, q \in S^1$ is defined as

$$\mathrm{dist}_{S^1}(q,p) = \min(|q-p|, 1-|q-p|).$$

3) Real projective space, $\mathbb{R}P^3$: it can be represented by three-dimensional sphere embedded in $\mathbb{R}^4$ with antipodal points identified. That is, $\mathbb{R}P^3 = S^3/x \sim -x$, in which $S^3 = \{x \in \mathbb{R}^4 \mid ||x|| = 1\}$.

Each element $x = (x_1, x_2, x_3, x_4) \in \mathbb{R}P^3$ is a unit quaternion, $x_1 + x_2\mathrm{i} + x_3\mathrm{j} + x_4\mathrm{k}$, representing a 3D rotation. The metric for two points $x, y \in \mathbb{R}P^3$ is defined as the length of the arc between these two points on the surface of the sphere

$$\mathrm{dist}_{\mathbb{R}P^3}(x,y) = \min(\cos^{-1}(x \cdot y), \cos^{-1}(x \cdot (-y))),$$

in which $(x \cdot y)$ denotes the dot product for vectors in $\mathbb{R}^4$.

Note: Sometimes Euler angles are used for representing 3D rigid rotations instead of quaternions. In this case, each rotation is represented as a vector $(x_1, x_2, x_3), x_i \in [-\pi, \pi]/ -\pi \sim \pi$. Since the topology of the space is $S^1 \times S^1 \times S^1$, the techniques described in the following sections can be used for Euler angles representation as well as quaternions.

4) Cartesian products of the spaces above: Given two metric spaces, $(T_1, \mathrm{dist}_{T_1})$ and $(T_2, \mathrm{dist}_{T_2})$, the weighted metric for two points, q, p, in the Cartesian product $T_1 \times T_2$ is defined as

$$\mathrm{dist}_{T_1 \times T_2}(q,p) = \sqrt{\mu_{T_1}\mathrm{dist}^2_{T_1}(q,p) + \mu_{T_2}\mathrm{dist}^2_{T_2}(q,p)},$$

in which the weights, μ_{T_1} and μ_{T_2}, are arbitrary nonzero real constants.

B. Problem Formulation

Consider one of the metric spaces described in Section II-A, $T = T_1 \times \cdots \times T_m$, in which each T_i is one of $\mathbb{R}$, S^1 or $\mathbb{R}P^3$. Consider the weighted metric defined on this manifold, $\mathrm{dist}_T : T \times T \to \mathbb{R}$. Suppose that a set of n data points, S, is a subset of T. The problem is: given any query point $q \in T$, efficiently report the point $p \in S$ that is closest to q.

Note that the brute-force computations of all the distances is one way of finding a correct nearest neighbor. However, our goal is to achieve significantly faster running times. We allow some preprocessing time for organizing the data points in a data structure. In return, we expect that the answer to the nearest-neighbor query is found significantly faster than the brute-force computations.

III. Nearest-Neighbor Searching Overview

There has been a significant interest in nearest-neighbor and related problems over the last couple of decades. For Euclidean data sets kd-tree-based methods proved to be one of the most effective in practice. The kd-tree data structure is based on recursively subdividing the rectangle enclosing the data points into subrectangles using alternating axis-aligned hyperplanes. Given the appropriate distance measure between points and rectangles in the space, kd-trees allow eliminate some of the points in the data set from the search during the query phase. Given a query point, q, it may be possible to discard some of the points in the data set based only on the distance between their enclosing rectangle and the query point. That is, based on one metric computation, the whole set of points inside the rectangle is eliminated from the search. The classical kd-tree uses $O(dn \lg n)$ precomputation time, and answers orthogonal range queries in time $O(n^{1-1/d})$. One of the first appearances of the kd-tree is in [13], and a more modern introduction appears in [11]. Improvements to the data structure and its construction algorithm in the context of nearest-neighbor searching are described in [28]. In [4] it is shown that using kd-trees for finding approximate nearest neighbors allows significant improvement in running time with a very small loss in performance for higher dimensions. Other data structures for nearest-neighbor searching in Euclidean spaces are used for high-dimensional problems [15], and for dynamic data [1].

Different techniques have been developed for nearest-neighbor searching in general metric spaces [10], [14]. Many efficient algorithms [8], [19], [7] were implemented and tested on various data sets [9], [7]. Most of these techniques consider the metric as a "black box" function provided to the algorithm. Usually these methods group the points in such a way that it is possible to eliminate some groups of points from the search in the query phase based on some inexpensive test. In the way, this approach is similar to kd-tree-based approach, in which the points are eliminated from the search if they are enclosed by a rectangle far enough from the query point. However, since these techniques are more general and allow any metric space to be searched, they are usually not as efficient on Euclidean spaces as techniques designed primarily for Euclidean spaces, such as kd-trees [9].

The goal of this paper is to show how to adapt kd-trees to handle spaces described in Section II, introducing only a little computational overhead for handling topological constraints and, therefore, keeping the simplicity and efficiency of kd-trees. Next Section introduces our method.

IV. Approach Based on KD-Trees

First, we elaborate on possible ways of using kd-trees for given spaces, and then we present our approach.

A. A Naive Way to Use Kd-Trees

To apply kd-tree-style reasoning to the metric spaces of interest, a naive approach would be to embed a given manifold into a higher-dimensional Euclidean space, and then treat the set of points lying on this manifold as a Euclidean data set. For example, the set of all rotations can be represented using 3×3 matrices, which places them in Euclidean space $\mathbb{R}^9$. The drawback of this approach is that the dimensionality of the space is significantly increased, which often implies worse performance of nearest neighbor methods. Moreover, the Euclidean metric in the resulting Euclidean space is different from the natural metric defined over quaternions. For many applications this is not tolerable, and kd-trees cannot be immediately applied. Next we show how a different approach can be taken so that the kd-tree data structure is adapted naturally and efficiently to the metric spaces of interest.

B. Representing the Spaces of Interest

Consider the metric spaces of interest before the identifications are done. That is, the circle is considered as a unit interval in $\mathbb{R}^1$ and the quaternion real projective space as a 3D sphere embedded in $\mathbb{R}^4$. The kd-tree can be first constructed inside $\mathbb{R}^1$ and $\mathbb{R}^4$. Next, to obtain a correct answer to the nearest-neighbor query, identifications and the correct metric are used in the query phase. That is, when computing distances from the query point to a point or an enclosing rectangle of a set of points, the correct metric respecting the topology of the space is used. In this manner, a rectangular decomposition is done on these non-Euclidean spaces, and, at the same time, the correct metric is used throughout the search.

In the rest of this subsection we define the notion of enclosing rectangle, and distance between a point and a rectangle in each of the defined metric spaces.

1) Euclidean one-space: The enclosing rectangles are regular intervals in $\mathbb{R}^1$, and the distance between a point, p, and a rectangle, $[a,b]$, is the usual Hausdorff metric:

$$\mathrm{dist}_{\mathbb{R}}(p,[a,b]) = \inf_{r\in[a,b]} \mathrm{dist}_{\mathbb{R}}(p,r).$$

2) Circle S^1: The enclosing rectangle for a set of points on the circle is any subinterval of $[0,1]$. The distance between a point, p, and a rectangle, $[a,b]$, is the Hausdorff distance on S^1:

$$\mathrm{dist}_{S^1}(p,[a,b]) = \inf_{r\in[a,b]} \mathrm{dist}_{S^1}(p,r).$$

3) Real projective space $\mathbb{R}P^3$: Rectangles that enclose the data lying on the unit sphere $S^3 \subseteq \mathbb{R}^4$ are usual rectangular regions $[a_1,b_1]\times\cdots\times[a_4,b_4]$ in $\mathbb{R}^4$. The distance between a point, p, and a rectangle, R, could be defined as the Hausdorff distance between p and the intersection of R with the surface of the sphere. However, the distance that we use in this paper is more efficient to compute and guarantees the correctness of the nearest-neighbor search, as we prove in Section IV-G. Essentially, the following is the Hausdorff distance between p and R in $\mathbb{R}^4$, respecting the identifications of $\mathbb{R}P^3$:

$$\mathrm{dist}_{\mathbb{R}P^3}(p,R) = \min(\mathrm{dist}_{\mathbb{R}^4}(p,R), \mathrm{dist}_{\mathbb{R}^4}(-p,R)).$$

4) Cartesian product of the spaces above: Consider the topological space T, such that T is a Cartesian product $T = T_1\times\cdots\times T_m$ of copies of $\mathbb{R}$, S^1 and $\mathbb{R}P^3$. Enclosing rectangles for this space are those formed by enclosing rectangles in the projections of T on each of $\mathbb{R}$, S^1 and $\mathbb{R}P^3$. The distance between a point and a rectangle is defined as

$$\mathrm{dist}_T(p,R) = \sqrt{\sum_i \mu_{T_i}\mathrm{dist}^2_{T_i}(p,R)}.$$

C. KD-Trees for the Spaces of Interest

The kd-tree-based approach for the nearest-neighbor problem formulated in Section II consists of first precomputing the data structure for storing points, and then searching this data structure when a query is given. In this subsection, we describe the kd-tree data structure for the manifolds of interest in more detail, and in the following subsections we provide the algorithms for the construction and query phases.

Consider the set of data points, S, lying inside a d-dimensional enclosing rectangle as described above. We build the kd-tree data structure inside this rectangle, and define it recursively as follows. The set of data points is split into two parts by splitting the rectangle that contains them into two child rectangles by a hyperplane, according to some specified splitting rule; one subset contains the points in one child box, and another subset contains the rest of the points. The information about the splitting hyperplane and the boundary values of the initial box are stored in the root node, and the two subsets are stored recursively in the two subtrees. When the number of the data points contained in some box falls below a given threshold, a node associated with this box is called

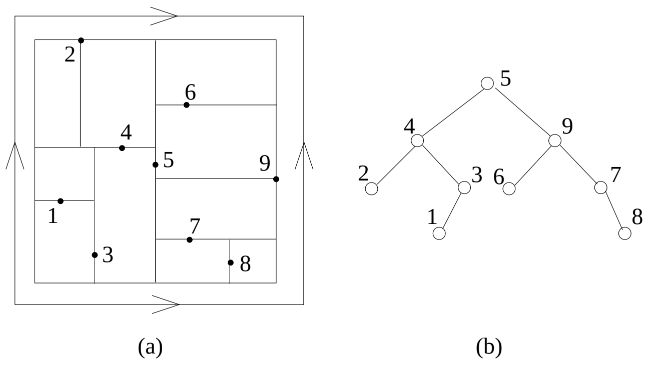


Fig. 2. A kd-tree: a) how a torus is subdivided, b) the corresponding binary tree.

a leaf node, and a list of coordinates for these data points is stored in this node.

We use splitting rules suggested in [26], which divide the current cell through its midpoint orthogonal to its longest side. If there are ties, it selects the dimension with the largest point spread. However, in the case in which points are all on one side of the splitting plane, the algorithm slides the plane toward the first encountered data point. According to [26] these rules perform very well with typical data sets in $\mathbb{R}^d$.

Figure 2 illustrates how the splitting is done, and how the corresponding binary tree looks for the data points on a torus.

D. Construction Phase

Our kd-tree is constructed using a recursive procedure, which returns the root of the kd-tree (see Figure 3). This construction algorithm is essentially identical to the case of constructing a kd-tree in a Euclidean space [26]. The identifications and proper metrics are not used in construction phase, and the points are treated as lying inside some $\mathbb{R}^d$ as described in the beginning of this Section.

E. Query Phase

The query phase must be handled differently in comparison to a standard kd-tree, by incorporating the correct metrics defined in Sections II and IV when traversing the tree. In everything else, the search proceeds in the same manner as the search in classical kd-trees. At first, the query algorithm descends to a leaf node that contains the query point, finds all distances from the data points in this leaf to the query point, and picks up the closest one. It then recursively visits only those surrounding rectangles that are closer to the query point than the closest point found so far (with respect to the correct metric). Those that are further are discarded from consideration. Figure 4 describes the query algorithm.

We borrowed some efficient techniques from [3] to further speed up the computations. Using squared distances prevents calculating costly square roots. We also modified method of *incremental distance calculation* for speeding up the calculations of a distance between the query point and a rectangle. This method can be described as follows. Let T be a Cartesian product of several manifolds, $T = T_1 \times \cdots \times T_m$, and let

```
BUILD_KD_TREE($P$, $d$, $T$, $m$, $b$, $s$)
Input: A set of points, $P$, the dimension of the space, $d$, the
topology of the space, $T$, the number of points to store in a
leaf, $m$, the bounding box, $b$, for $P$, and the splitting rule, $s$.
Output: The root of a kd-tree storing $P$
1   if $P$ contains less than $m$ points
2       then return a Leaf storing these points
3       else split $b$ into two subboxes, $b_1$, $b_2$, according
              to $s$ by plane $l$, orthogonal to dimension $k$.
4             Find $P_1$ and $P_2$, the sets of the data points
              falling into boxes $b_1$ and $b_2$.
5             $v_1$ =BUILD_KD_TREE($P_1, d, T, m, b_1, s$)
6             $v_2$ =BUILD_KD_TREE($P_2, d, T, m, b_2, s$)
7             Create a Node $v$ storing the splitting plane, $l$,
              the splitting dimension, $k$, the topology of the
              space $T_K$ of this dimension, the projection
              of the box, $b$, on $T_K$, and $v_1$ and $v_2$, the
              children of $v$.
8             return $v$
```

Fig. 3. The algorithm for constructing kd-tree in topological space T.

coordinate axis k correspond to some space T_K. Suppose that a query point, q, and an enclosing rectangle for the data set, S, in T are given. Divide R with a plane orthogonal to coordinate axis k into two child rectangles R_1 and R_2. If it is known that $dist^2(q, R) = d_{box}$, then the squared distance from one of the rectangles (without loss of generality it can be R_1) to q is also d_{box}. To calculate $dist^2(q, R_2)$, note that R_2 has the same projections as R on every T_i except for T_K, by definition of weighted metric in T (Section IV-B.4). Therefore, if $dist^2_{T_K}(q, R_1) = dist^2_{T_K}(q, R)$, then

$$dist^2_T(q, R_2) = d_{box} - dist^2_{T_K}(q, R_1) + dist^2_{T_K}(q, R_2).$$

Therefore, calculating distance from a point to a rectangle node in d-dimensional space, T, takes $O(d)$ time only for the root node, and for any other node the time is proportional to the time for calculating distance from a point to a rectangle in T_K, the subspace of T.

F. Making KD-Trees Dynamic

In some algorithms, such as the RRT, the number of points grows incrementally while nearest-neighbor queries are performed at each iteration. In this case, it is inefficient to rebuild the kd-tree at every iteration. One approach to make the nearest-neighbor algorithm dynamic is to use the point insertion operation with tree rebalancing [27]. It is costly, however, to ensure that the trees are balanced.

Another approach, which we used in our implementation, is a standard method to perform static-to-dynamic transformation of a data structure, called *the logarithmic method* [6]. For n points, there is a tree that contains 2^i points for each "1" in the i^{th} place of the binary representation of n. As bits are cleared in the representation due to increasing n, the trees are deleted, and the points are included in a tree that corresponds to the higher-order bit which changed to "1". This general scheme incurs logarithmic-time overhead, regardless of dimension. It

```
KDTree::SEARCH($q$)
Output: the closest to $q$ point $p$ stored in kd-tree.
1  Calculate squared distance $d_{box}$ from the box
   associated with the $root$ node to $q$.
2  $p = NULL$
3  $root \rightarrow$ SEARCH($d_{box}$, $\infty$, $p$)
4  return $p$
```

```
Node::SEARCH($d_{box}$, $d_{best}$, $p$)
Input: squared distance, $d_{box}$, from $q$ to the box containing
current Node, and squared distance, $d_{best}$, from $q$ to the
closest point, $p$, seen so far; $d_{best}$ and $p$ are to be updated.
1  if $d_{box} < d_{best}$
2      Split $b_K$ (the projection of the current Node onto
       the space $T_K$, stored in this Node) into two
       subboxes, $b_{K_1}$ and $b_{K_2}$, by the splitting line $l$,
       corresponding to $v_1$ and $v_2$ respectively.
3      $d_1 = dist^2_{T_K}(q, b_{K_1})$
4      $d_2 = dist^2_{T_K}(q, b_{K_2})$
5      if $d_1 < d_2$
6          then $v_1 \rightarrow$ SEARCH($d_{box}$, $d_{best}$, $p$)
7               $v_2 \rightarrow$ SEARCH($d_{box} - d_1 + d_2$, $d_{best}$, $p$)
8          else $v_2 \rightarrow$ SEARCH($d_{box}$, $d_{best}$, $p$)
9               $v_1 \rightarrow$ SEARCH($d_{box} - d_2 + d_1$, $d_{best}$, $p$)
```

```
Leaf::SEARCH($d_{box}$, $d_{best}$, $p$)
Input: squared distance, $d_{box}$, from $q$ to the box containing
the current Leaf, and squared distance, $d_{best}$, from $q$ to the
closest point, $p$, seen so far; $d_{best}$ and $p$ are to be updated.
1  Calculate squared distances from $q$ to all the points
   in the current Leaf, and update $p$ and $d_{best}$.
```

Fig. 4. The algorithm portions for searching kd-tree on the root level and internal and leaf nodes levels.

is also straightforward to implement, and leads to satisfactory experimental performance.

G. Analysis

Proposition 1 *The algorithm presented in Figure 4 correctly returns the nearest neighbor.*

Proof: We argue that the points in the kd-tree, that are not visited by our algorithm cannot be the closest neighbors to the query point, since they are always further from the query point than some point which was already visited by the algorithm. At first, the search procedure descends to the leaf node to which the query point belongs. Therefore, the closest point to the query point from this leaf will be the first candidate to be the nearest neighbor. After searching this leaf node the algorithm skips only those nodes (enclosing rectangles) that are further from the query point than the candidate to the nearest neighbor seen so far. Any point inside a node that was skipped cannot be the nearest neighbor, since the point inside a rectangle is further from the query point than the rectangle itself. This holds true for Hausdorff distances defined on $\mathbb{R}^1$ and S^1 by definition. It is also true for the distance we used on $\mathbb{R}P^3$, since

$$\mathrm{dist}_{\mathbb{R}P^3}(q, R) \leq \mathrm{dist}_{\mathbb{R}^4}(q, p),$$

for all $p \in R$, by definition of $\mathrm{dist}_{\mathbb{R}P^3}(q, R)$. Since the length of the arc along the sphere is longer than the length of the corresponding chord, we obtain

$$\mathrm{dist}_{\mathbb{R}P^3}(q, R) \leq \mathrm{dist}_{\mathbb{R}^4}(q, p) \leq \mathrm{dist}_{\mathbb{R}P^3}(q, p),$$

for all p $\in R$. ■

Proposition 2 *For n points in dimension d, the construction time is $O(dn \lg n)$, the space is $O(dn)$, and the query time is logarithmic in n, but exponential in d.*

Proof: This follows directly from the well-known complexity of the basic kd-tree [13]. Our approach performs correct handling of the topology without any additional asymptotic complexity. ■

The metric evaluations are more costly due to identifications in the manifold definition; however, this results only in a larger constant in the asymptotic analysis. For example, each S^1 subspace requires two more operations per distance computation (see Sections II-A.2, IV-B.2), which essentially does not affect the overall running time. For each $\mathbb{R}P^3$ there are sometimes 3 to 6 additional operations per distance computation (see Sections II-A.3, IV-B.3). Two of these operations are $\cos^{-1}$, which are expensive and sometimes take several orders of magnitude longer than basic addition or multiplication operations. This results in higher constants in the asymptotic running time for spaces containing $\mathbb{R}P^3$.

By following several performance enhancements recommended in [26], the effects of high dimensionality on the query time are minimized, yielding good performance for nearest-neighbor searching in up to several dozen dimensions in both ANN and our algorithm. Performance can be further improved by returning approximate nearest neighbors, if suitable for a particular motion planning method.

V. Experiments

We have implemented our nearest-neighbor algorithm in C++ as part of the new library, MPNN, for nearest-neighbor searching on dynamic data sets in the context of motion planning. This library is publicly available at [29]. The kd-tree implementation that we used in MPNN is borrowed from the ANN library. We then used MPNN in implementations of RRT-based and PRM-based planners in the Motion Strategy Library [21]. The experiments reported here were performed on a 2.2 GHz Pentium IV running Linux and compiled under GNU C++.

We have compared the performance of the MPNN library to the brute-force algorithm as well as two general metric space nearest-neighbor libraries, cover trees [7] and sb(S) [9]. Figures 5-7 indicate the performance of these methods in the time to complete 100 queries in various topological spaces. The performance improvement of the kd-tree-based approach is several orders of magnitude in some cases over other methods. As the dimension of the space increases, though, the brute-force algorithm outperforms all the methods, as well as kd-trees, because of the hidden exponential dependencies on the dimension in these methods. However, the data sets in motion planning often have small intrinsic dimensionality. Obstacles and other constraints, such as kinematic or differential constraints, reduce the effective dimension of the

$\mathbb{R}^d$				
d	MPNN	naive	cover tree	sb(S)
3	0.2 + 0.01	0.22	0.63 + 0.01	2.43 + 0.02
6	0.34 + 0.01	0.38	0.79 + 0.1	14.0 + 0.08
9	0.48 + 0.02	0.56	1.36 + 0.19	41.7 + 0.46
12	0.64 + 0.14	0.74	3.72 + 0.77	90.6 + 0.91
15	0.77 + 0.47	0.94	9.52 + 1.30	158.3 + 1.76
18	0.97 + 1.61	1.14	23.0 + 2.16	238.3 + 3.25
21	1.16 + 2.17	1.55	51.5 + 2.36	273.4 + 3.19
24	1.40 + 3.20	1.65	84.4 + 3.31	327.6 + 5.71
27	1.58 + 3.70	1.78	135.4 + 3.59	373.3 + 6.59
30	1.80 + 4.11	2.04	242.0 + 4.33	392.8 + 7.59

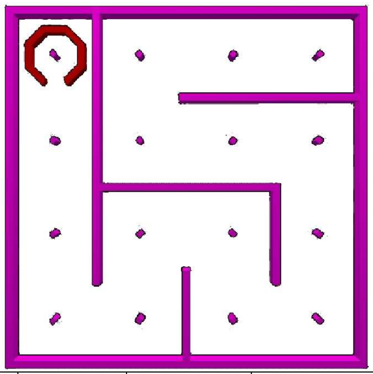
Fig. 5. Nearest-neighbor computations are shown for 50000 data points generated at random in Euclidean spaces $\mathbb{R}^d$. The time to perform 100 queries is shown for the naive, brute-force algorithm. For other methods the construction time is added to the time required to perform 100 queries.

$(S^1)^d$				
d	MPNN	naive	cover tree	sb(S)
3	0.21 + 0.01	0.26	0.69 + 0.02	2.97 + 0.01
6	0.32 + 0.01	0.44	1.06 + 0.04	13.0 + 0.07
9	0.48 + 0.01	0.69	2.34 + 0.15	39.8 + 0.46
12	0.63 + 0.05	0.85	7.31 + 0.87	88.5 + 1.06
15	0.77 + 0.38	1.04	17.8 + 2.11	156.9 + 1.77
18	0.98 + 0.89	1.21	45.4 + 2.89	239.7 + 2.9
21	1.20 + 1.83	1.37	104.8 + 3.73	321.7 + 3.26
24	1.40 + 2.68	1.58	186.9 + 6.21	411.0 + 8.83
27	1.62 + 3.30	1.72	289.7 + 5.62	478.5 + 9.07
30	1.80 + 4.03	1.89	545.6 + 8.43	499.2 + 9.25

Fig. 6. Nearest-neighbor computations are shown for 50000 data points generated at random in spaces $(S^1)^d$. The time to perform 100 queries is shown for the naive, brute-force algorithm. For other methods the construction time is added to the time required to perform 100 queries.

$(\mathbb{R}^3 \times \mathbb{R}P^3)^k$					
d	k	MPNN	naive	cover tree	sb(S)
7	1	0.29 + 0.02	3.17	1.1 + 0.06	19.5 + 0.1
14	2	0.47 + 0.12	6.18	5.6 + 0.4	106 + 0.9
21	3	0.75 + 1.10	9.20	25.8 + 2.8	231 + 1.7
28	4	0.89 + 1.82	12.2	89.5 + 4.3	457 + 5.9
35	5	0.97 + 3.72	15.2	215 + 6.8	723 + 10.1
42	6	1.17 + 6.2	18.2	469 + 9.2	981 + 18.5
49	7	1.43 + 9.32	20.9	658 + 12.0	1205 + 21
56	8	1.63 + 11.2	24.0	1435 + 13.4	1374 + 27

Fig. 7. Nearest-neighbor computations are shown for 50000 data points generated at random in spaces $(\mathbb{R}^3 \times \mathbb{R}P^3)^k$. The time to perform 100 queries is shown for the naive, brute-force algorithm. For other methods the construction time is added to the time required to perform 100 queries.



$\mathbb{R}^2 \times S^1$	MPNN	naive	cover tree	sb(S)
nodes	29,754	30,805	31,712	30,277
time (sec)	99.47	7,496.41	353.37	175.94

Fig. 8. This example involves moving the C-shaped object to the end of the maze. There are many narrow corridors in the configuration space, $\mathbb{R}^2 \times S^1$. The problem was solved using a RRTConCon [25].

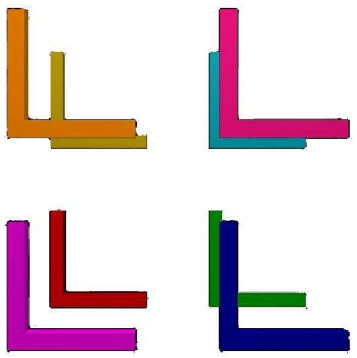
problem. Our experiments (see Figure 7) also suggest that kd-trees outperform other methods in up to 56 dimensions on randomly generated sets in $(\mathbb{R}^3 \times \mathbb{R}P^3)^k$ due to the choice of the constants in weighted metric. Rotations are usually given smaller weight than translations in motion planning problems. For example, $\mu_{\mathbb{R}^3} = 1, \mu_{\mathbb{R}P^3} = 0.15$ are the standard values used in MSL. This works to the advantage of kd-tree-based approaches, and, therefore, makes them potentially applicable to many motion planning problems with high-dimensional configuration spaces.

Figures 8 and 9 show performance of the methods in bidirectional RRT-based planners for 3-dof and 48-dof problems, respectively. Performance for a basic PRM applied to a 6-dof example is shown in Figure 10. These experiments suggest that the MPNN library can be effectively used in up to 56-dimensional spaces with considerable running time improvements in the performance of RRT-based and PRM-based planning algorithms. It is important to note, however, that nearest-neighbor searching does not represent the only bottleneck in motion planning. Sampling strategies and collision detection issues are also critical. For the experiments, we focused on examples that lead to a large number of nodes so that the nearest-neighbor searching would dominate. In general, the development of the most efficient algorithms should involve consideration of all of these issues.

VI. Conclusions

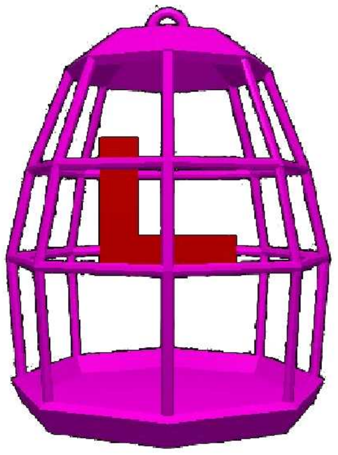
We have presented and implemented a practical algorithm for performing efficient nearest-neighbor search for the topological spaces that commonly arise in motion planning. We have illustrated the importance of performing efficient nearest-neighbor computations in the context of path planning. Our method extends previous techniques designed for Euclidean spaces by building kd-trees that respect topological identifications and the resulting distance metric.

Our method has been implemented, and is observed to be orders of magnitude faster than naive nearest-neighbor searching. It is substantially faster even in high-dimensional spaces, which are of great importance in motion planning. We evaluated the implemented algorithm as a means to accelerate performance in both PRM and RRT algorithms. Substantial improvement was observed in both cases; however, it is important to note that the benefits are substantial only if the nearest-neighbor computations dominate the total running



$(\mathbb{R}^3 \times \mathbb{R}P^3)^8$	MPNN	naive	cover tree	sb(S)
nodes	18,920	18,485	20,907	22,210
time (sec)	2,055.57	4,152.19	5,273.14	6,161.47

Fig. 9. This 56-dimensional problem involves exchanging positions of 8 L-shaped objects contained in a rectangular box. It was solved using RRTConCon [25].



$\mathbb{R}^3 \times \mathbb{R}P^3$	MPNN	naive	cover tree	sb(S)
nodes	37,634	37,186	35,814	37,922
time (sec)	191.96	2,302.49	187.99	361.43

Fig. 10. This example is solved using the PRM approach [17]. The goal is to move the 3D rigid object out of the cage.

time. Collision detection is a competing bottleneck in path planning algorithms; therefore, strong performance benefits can be expected in cases in which the number of PRM or RRT nodes is large in comparison to the number of primitives in the geometric models used for collision detection.

Several directions are possible for future work. The extension to different topological spaces can also be applied to other extensions of the kd-tree that have been used for nearest-neighbor searching, such as the relative neighbor graph [3] and balanced box-decomposition tree [5]. It has been shown recently that it is possible to remove exponential dependencies in dimension from the nearest-neighbor problem [16], [18]. Powerful new techniques are based on approximate distance-preserving embeddings of the points into lower-dimensional spaces. It remains to be seen whether these theoretical ideas will lead to practical algorithms, and whether they will yield superior performance for the dimension and number of points that are common in path planning problems. In path planning problems that involve differential constraints (nonholonomic and kinodynamic planning), it might be preferable to use complicated distance functions [12]. Such functions should be well-suited for a particular nonlinear system, and they might not even be symmetric. In these difficult cases, it remains to determine practical, efficient nearest-neighbor algorithms.

References

[1] P. K. Agarwal, L. J. Guibas, H. Edelsbrunner, J. Erickson, M. Isard, S. Har-Peled, J. Hershberger, C. Jensen, L. Kavraki, P. Koehl, M. Lin, D. Manocha, D. Metaxas, B. Mirtich, D. Mount, S. Muthukrishnan, D. Pai, E. Sacks, J. Snoeyink, S. Suri, and O. Wolefson. Algorithmic issues in modeling motion. *ACM Comput. Surv.*, 34(4):550–572, 2002.

[2] N. M. Amato and Y. Wu. A randomized roadmap method for path and manipulation planning. In *IEEE Int. Conf. Robot. & Autom.*, pages 113–120, 1996.

[3] S. Arya and D. M. Mount. Algorithm for fast vector quantization. In *IEEE Data Compression Conference*, pages 381–390, March 1993.

[4] S. Arya and D. M. Mount. Approximate nearest neihgbor queries in fixed dimensions. In *ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280, 1993.

[5] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions (revised version). In *Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, January 1994.

[6] J.L. Bentley and J. Saxe. Decomposable searching problems I: Static to dynamic transformation. *J. Algorithms*, 1:301–358, 1980.

[7] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. 2004. http://www.cs.rochester.edu/∼beygel/cover_tree.ps.

[8] K. L. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Comput. Geom.*, 22:63–93, 1999.

[9] K. L. Clarkson. Nearest neighbor searching in metric spaces: Experimental results for sb(s). 2003. http://cm.bell-labs.com/who/clarkson/Msb/readme.html.

[10] K. L. Clarkson. Nearest-neighbor searching and metric space dimension. 2005. http://cm.bell-labs.com/who/clarkson/nn_survey/b.pdf.

[11] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, 1997.

[12] E. Frazzoli, M. A. Dahleh, and E. Feron. Robust hybrid control for autonomous vehicles motion planning. Technical Report LIDS-P-2468, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, 1999.

[13] J. H. Friedman, J. L. Bentley, and R.A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[14] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Trans. Database Syst.*, 28(4):517–580, 2003.

[15] P. Indyk. Nearest neighbors in high-dimensional spaces. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry, chapter 39*. CRC Press, 2004. 2rd edition.

[16] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 604–613, 1998.

[17] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Trans. Robot. & Autom.*, 12(4):566–580, June 1996.

[18] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *ACM Symposium on Theory of Computing*, pages 599–608, May 1997.

[19] R. Krauthgamer and J. R. Lee. Navigating nets: Simple algorithms for proximity search. In *Proc. Fifteenth Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 798–807, 2004.

[20] J. J. Kuffner and S. M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 995–1001, 2000.

[21] S. M. LaValle. MSL: Motion strategy library. http://msl.cs.uiuc.edu/msl/.

[22] S. M. LaValle. Rapidly-exploring random trees: A new tool for path planning. TR 98-11, Computer Science Dept., Iowa State University, Oct. 1998.

[23] S. M. LaValle. *Planning Algorithms*. [Online], 2004. Available at http://msl.cs.uiuc.edu/planning/.

[24] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 473–479, 1999.

[25] S. M. LaValle and J. J. Kuffner. Rapidly-exploring random trees: Progress and prospects. In *Workshop on the Algorithmic Foundations of Robotics*, 2000.

[26] D. M. Mount. ANN programming manual. Technical report, Dept. of Computer Science, U. of Maryland, 1998.

[27] M. H. Overmars and J. van Leeuwen. Dynamic multidimensional data structures based on Quad- and K-D trees. *Acta Informatica*, 17:267–285, 1982.

[28] R. L Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6:579–589, 1991.

[29] A. Yershova. MPNN: Nearest neighbor library for motion planning. 2005. http://msl.cs.uiuc.edu/∼yershova/MPNN/MPNN.htm.